**STUDENT NAME:** _____ **STUDENT ID:** _____

# CS 222/122C – Fall 2016, Final Exam

**Principles of Data Management**
**Department of Computer Science, UC Irvine**
**Prof. Chen Li**
**(Max. Points: 100)**

**Instructions:**

- **This exam has seven required questions and one extra-credit question.**
- **This exam is closed book. However, you can use one cheat sheet (A4 size).**
- **The total time for the exam is 120 minutes, so budget your time accordingly.**
- **Be sure to answer each part of each question after reading the whole question carefully.**
- **If you don't understand something, ask for clarification.**
- **If you still find ambiguities in a question, write down the interpretation you are taking and then answer the question based on that interpretation.**

| QUESTION | POINTS | SCORE |
|---|---|---|
| 1 | 12 | |
| 2 | 16 | |
| 3 | 11 | |
| 4 | 13 | |
| 5 | 16 | |
| 6 | 12 | |
| 7 | 20 | |
| TOTAL | 100 | |
| Extra credits | 15 | |

**Question 1 (12 points)**

a) **(3 pts)** Briefly explain what is schema versioning and how it works in relational DBMS.

Schema versioning
adding/dropping attributes

b) **(3 pts)** What are the three basic functions that should be implemented by each operator in a query plan?

init(), getNext(), close()

c) **(3 pts)** What are composite index keys? Give one query example where composite index keys can be useful for indexing and explain the reasons.

A key that contains several fields is called a composite key. (1 pts)

If we have an index on the composite key: <E.dno, E.sal>, the following two queries can benefit from it:

(1) **SELECT** E.dno, **MIN** (E.sal)
**FROM** Emp E
**GROUP BY** E.dno … index-only query

(2) **SELECT** E.dno
**FROM** Emp E
**WHERE** E.dno = n **AND** E.sal = s … more efficient query

d) **(3 pts)** Given two bags R = {'a', 'b', 'b', 'c', 'd'} and S = {'b', 'b', 'e', 'f', 'f'}, write down the results of their **UNION**, **UNION ALL**, **INTERSECT**, and **EXCEPT** separately.

**UNION:** {'a', 'b', 'c', 'd', 'e', 'f'}
**UNION ALL:** {'a', 'b', 'b', 'b', 'b', 'c', 'd', 'e', 'f', 'f'}
**INTERSECT:** {'b'}

**Question 2 (16 points) External merge sort**

Suppose we have M = 10,000 records in a heap file. Each record is 120 bytes long. The page size is 4,096 bytes. Each page uses 2 bytes to store the free-space pointer, and 2 bytes to store the number of slots in the page. For each record, its slot in the directory uses 2 bytes its offset and 2 bytes for its length. We have B = 6 available in-memory buffer pages to sort the file using the external merge sorting algorithm covered in our lectures.

For each of the following questions, provide enough calculation details.

**Note:** Since the calculations of later subquestions depend on earlier answers, you should refer to the results of earlier questions as variables, and use them in later calculations. Then plug in those real values to compute the final results. You **may** get **partial** credits if your calculation is correct but using wrong earlier answers.

a) **(2 pts)** How many records are in each page?

Each page contains (4,096 - 4) / (120 + 4) = 33 records.

b) **(2 pts)** How many sorted runs are produced by Pass 0?

The number of pages of the file is: ceil(10,000 / 33) = 304.
The number of sorted runs after pass 0 is: ceil(304 / 6) = 51.

c) **(3 pts)** How many sorted runs are produced by Pass 1?

ceil(51 / 5) = 11

d) **(3 pts)** How many passes, including Pass 0, do we need to sort the file?

5^2 < 51 < 5^3 => 3 passes excluding Pass 0
3 + 1 = 4 passes

e) **(3 pts)** How many I/Os are required to sort the file, excluding the writes in the last pass?

(2 * 4 - 1) * 304 = 2128

f) **(3 pts)** Suppose double buffering is used in all passes except Pass 0. How many passes are needed to sort the file, including Pass 0?

2 ^ 5 < 51 < 2 ^ 6
6 + 1 = 7 passes

**Question 3 (11 points): Join**

Suppose you are in charge of the recruiting process of your company's summer internships. You have two tables as follows:

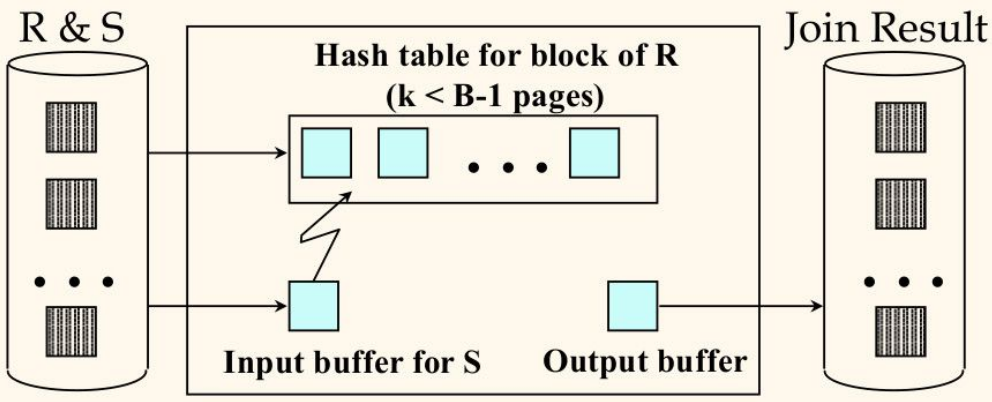
*Applicant(name, email, phone,* ***school_id****, GPA, ...)*: 100,000 records, 10 records per page.

*School(****school_id****, school_name, state, num_students, avg_GPA, ...)*: 1,000 records, 10 records per page.

Now you want to do a join *Applicant* $\bowtie_{Applicant.school\_id= School.school\_id}$ *School*. You have B = 22 available in-memory buffer pages for joining the tables.

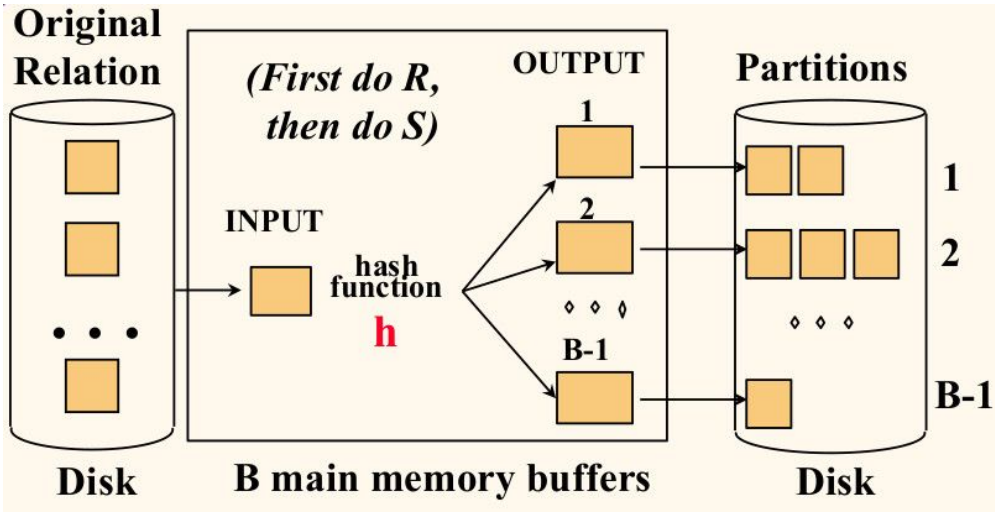
For each of the following questions, provide enough calculation process **with diagrams** to get full credits.

a) **(4 pts)** Choose a join order to minimize the number of disk I/Os of using a Block Nested Loops Join. For this order, calculate the number of disk I/Os.



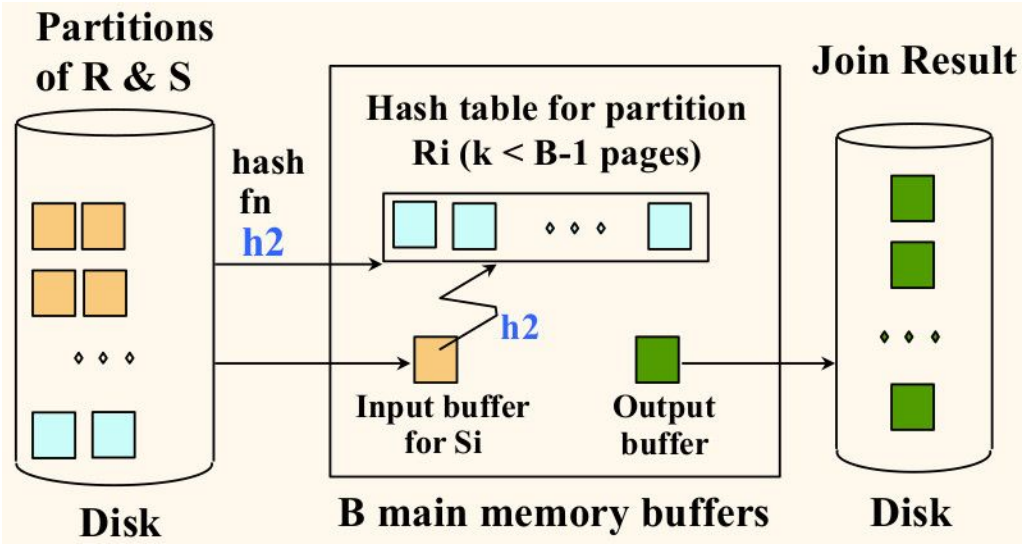

Applicant: 100,000 / 10 = 10,000 pages
School: 1,000 / 10 = 100 pages
100 + (100 / 20) * 10,000 = 50,100

b) **(4 pts)** How many page I/Os are required in the partitioning/building phase of Grace Hash Join?




100 / (22 - 1) < (22 - 2) => 1 pass of partitioning is enough
2 * (10,000 + 100) = 20,200

c) **(3 pts)** Which table should be used to build the in-memory hash table in the probing/matching phase of Grace Hash Join? Why?




School. It is smaller, thus it's likely that each partition of the table can be fit into the memory after only one pass of partitioning.

**Question 4 (13 points): Union**

Suppose your company's internship program receives electronic applications from a recruitment website and hardcopy applications from on-campus career fairs. Applications received from the two sources are organized into two separate tables as follows:

*Applicant1(name, email, phone, school_id, GPA, ...)*: 100,000 records, 10 records per page.
*Applicant2(name, email, phone, school_id, GPA, ...)*: 20,000 records, 10 records per page.

As students are desperate to get into your company, many of them turn in multiple applications through both sources. Now you want to generate a single table that contains information of all **unique** applicants by doing a **UNION** operation. You have 21 available in-memory buffer pages for the operation.

For each of the following questions, provide enough calculation process to get full credits.

**a) (5 pts)** What is the cost (number of disk I/Os) of the UNION operation if a sorting-based approach is used (e.g., sort both tables then merge the sorted tables)? Exclude the final write operations.

Applicant: ceil(10,000 / 21) = 477 runs
20^2 < 477 < 20^3
3 + 1 = 4 passes => 2 * 4 * 10,000 = 80,000
Applicant2: ceil(2,000 / 21) = 96 runs
20^1 < 96 < 20^2
2 + 1 = 3 passes => 2 * 3 * 2,000 = 12,000
80,000 + 12,000 + 10,000 + 2,000 = 104,000

**b) (5 pts)** What is the cost (number of disk I/Os) of the UNION operation if a hash-based approach is used? Exclude the final write operations.

5 * (10,000 + 2,000) = 60,000

**c) (3 pts)** Analyze the two methods above. For each of them, show at least one advantage and disadvantage compared to the other.

| | Pros | Cons |
|---|---|---|
| Sorting-based | in-order result | more disk I/Os |
| Hash-based | fewer disk I/Os | result is not ordered |

**Question 5 (16 points): Projection push down**

In the System-R query optimizer, suppose we want to implement a technique to push projection down to make a query plan more efficient. In this question, we want to go through the details of this technique starting with an example. Consider three tables with the following schemas:

R(A, B, C, D), S(D, E, F, G), T(G, H, I, J, K)

We are given a query on these tables:

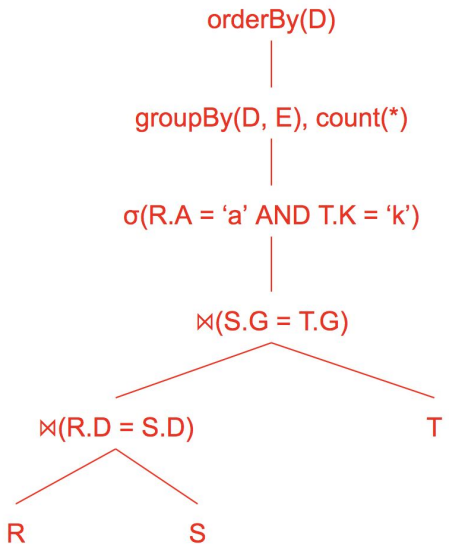
```
SELECT D, E, COUNT(*)
FROM R, S, T
WHERE R.D = S.D AND S.G = T.G
  AND R.A = 'a' AND T.K = 'k'
GROUP BY D, E
ORDER BY D
```

(notice that the in the original question, “order by B” is incorrect because B is not available after the group by aggregation. It is changed to “order by D”)

a) **(4 pts)** Draw a tree diagram to show a logical plan for this query, where a project operator is used only once.

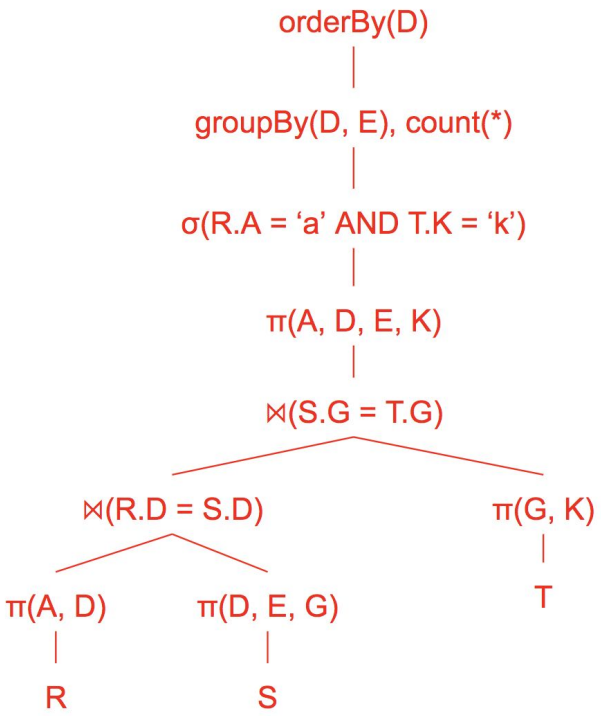
Notice that we do not need a projection operator because groupBy aggregation will drop all other attributes, and only keep the group by attributes, and aggregation result attribute. However, it will not be considered wrong if you add another projection operator.

This solution doesn’t perform selection pushdown. Other query plans that do some sort of selection pushdown optimization are also correct.



b) **(4 pts)** Transform the logical plan to another plan by introduction projection operators as deep (i.e., close to leaf nodes) as possible. Show the new logical plan, and briefly explain how you do the transformation.

This answer is based on the query plan in a). Without selection pushdown, some attributes can't be dropped early, so it might not be the most optimal plan overall.
As long as your answer is correct based on the query plan you gave in a), you will get full points.




c) **(4 pts)** Briefly explain the pros and cons of the new plan compared to the first one.

Pros: smaller records size => fewer disk I/Os
Cons: time to execute projection

d) **(4 pts)** Based on the observation in b), propose a generalized equivalence rule to transform one logical plan to another by introducing projection operators as early as possible. In particular, for each operator, discuss how to introduce a projection operator on top of it and what attributes can be included in the projection. Use a diagram to illustrate the rule.

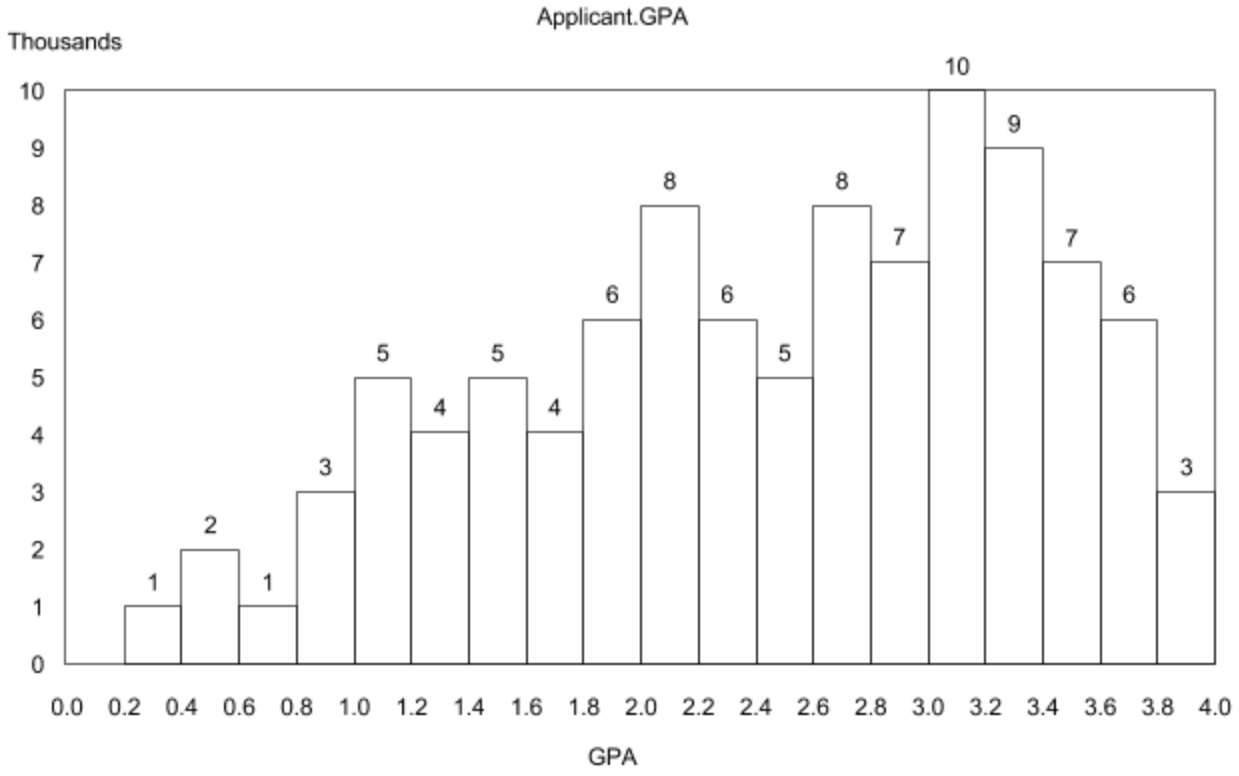
For each operator, examine all of its ancestors. Discard all fields that is not required by any of its ancestors.

**Question 6 (12 points): Cost estimation using histograms**

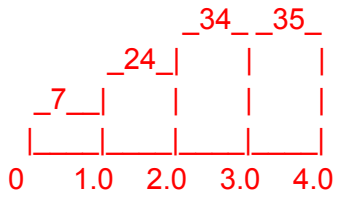
Consider the following table:

*Applicant(name, email, phone, school_id, GPA, ...)*: 100,000 records, 10 records per page.

The distribution of *GPA* is shown below.




a) **(2 pts)** Draw a 4-bucket **equi-width** histogram on Applicant.GPA.

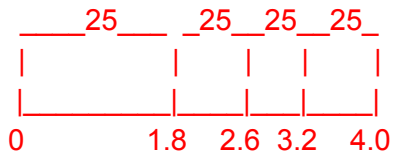



b) **(3 pts)** Using the **equi-width histogram above** to estimate the cardinality of the query:

**SELECT** *
**FROM** Applicant
**WHERE** Applicant.GPA >= 2.7 **AND** Applicant.GPA <= 3.6.

34,000 * 0.3 + 35,000 * 0.6 = 31,200

c) **(3 pts)** Draw a 4-bucket **equi-height** histogram on Applicant.GPA.




d) **(4 pts)** Based on the equi-height histogram above, choose a proper X value in the following query to return 300 records:

**SELECT** *
**FROM** Applicant
**WHERE** Applicant.GPA >= X **AND** Applicant.school_id <= 10.

Assume the *school_id* values range from 1 to 1,000 uniformly, and GPA and school_id are independent from each other.

100,000 * 10 / 1,000 * P = 300 => P = 0.3
(0.3 - 0.25) / 0.25 = 1 / 5
(3.2 - 2.6) / 5 = 0.12
3.2 - 0.12 = 3.08

**Question 7 (20 points): System-R optimizer**

We have a database with the following tables:

- Students(sid, name, email)
- Courses(cid, title, department, credits)
- Take(sid, cid, quarter)

Suppose we have the following indexes on the tables:

- Unclustered B+ tree on Students.sid
- Unclustered B+ tree on Courses.cid
- Unclustered B+ tree on Courses.department
- Unclustered B+ tree on Take.sid
- Unclustered B+ tree on Take.cid

Consider the following query:

```
SELECT sid, count(*)
FROM Students S, Take T, Courses C
WHERE S.sid = T.sid AND T.cid = C.cid
      AND C.department = 'CS' AND T.quarter = 'winter 2016'
GROUP BY sid;
```

We want to use the System-R optimizer to generate an efficient plan for this query.

a) **(5 pts)** For each base table, show all the access methods considered by the optimizer. For each relation and its access methods, indicate which of the methods will be kept for the next phase and explain why. Clearly mark those access methods with an interesting order.

- Students:
  - Full scan, kept because it's cheaper than B+ tree scan
  - B+ tree scan on sid (interesting order): kept
  - B+ tree search on sid for a constant: kept for a later index-based join
- Courses:
  - Full scan, not kept because it's worse than B+ tree search on department
  - B+ tree scan on cid (interesting order), kept
  - B+ tree search on department, kept because it's the cheapest
  - B+ tree search on cid for a constant: kept for a later index-based join
- Take:
  - Full scan, kept because it's cheaper than two B+ tree scans
  - B+ tree scan on cid (interesting order), kept
  - B+ tree search on cid for a constant: kept for a later index-based join
  - B+ tree scan on sid (interesting order), kept
  - B+ tree search on sid for a constant: kept for a later index-based join

b) **(5 pts)** Explain how the optimizer generates efficient access methods for joining the tables **Students** and **Take**. You need to explain the main idea of how the optimizer considers all possible access/join methods. You don't need to show all these enumerations.

1) Students join Take: For each access method on Students kept from the previous step, for each access method on Take kept from the previous step, consider all possible valid join methods: block nested loop join, index-based join, sort merge join, hash join, etc. Estimate the cost of each subplan. Use the interesting order from the previous method, if any, when estimating the cost of a sort-merge join. For each interesting order, select the join method with the smallest cost, and remove those subplans that are dominated by another subplan in terms of both cost and interesting order(s).
2) Repeat the same step for Take join Students;

c) **(5 pts)** Explain the remaining steps taken by the optimizer to generate the final physical plan. Make sure to include how the results of earlier steps are used, and how interesting orders are considered. Also explain how the "GROUP BY" operation is considered by the optimizer.

Pass 3:

(1) (Students, Take) JOIN Courses: Do the same as step 1) in Question b) to join (Students, Take) and Course.
(2) Do NOT consider Courses JOIN (Students, Take) since it's not a left-deep tree.
(3) Do the same for (Courses, Take) with Students.
(4) The previous step does not consider (Students, Courses) since it's a cross product.

Pass 4: For all the subplans from pass 3, consider various ways to do the group by, possibly using their interesting orders, and select the one with the lowest cost.

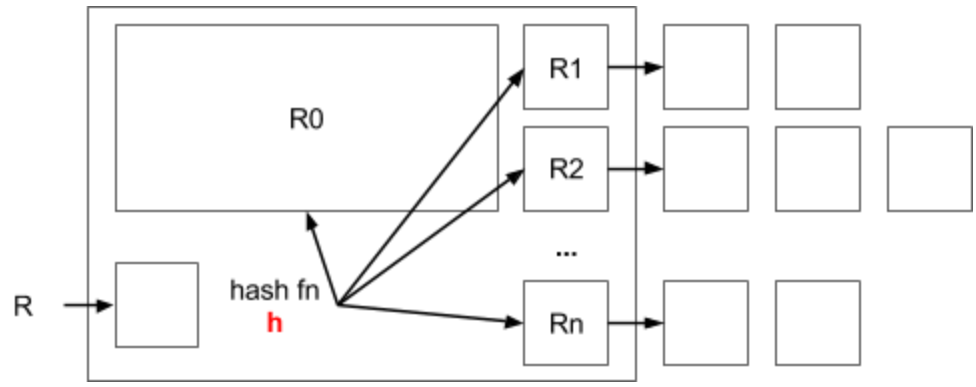
d) **(5 pts)** Using this example to briefly explain at least four main ideas in the System-R optimizer as covered in our lectures.

- Interesting orders
- Left-deep trees only
- Avoid Cartesian products
- Dynamic programming to do plan enumeration
- Selection push down
- Deal with group by at the end
- Deal with nested subqueries as blocks, and optimize them separately (not shown in the example)

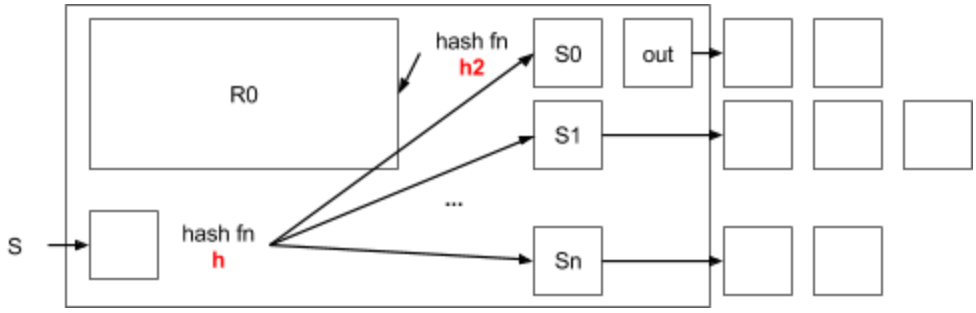
**Extra-Credit Question (15 points): Hybrid hash join**

a) **(6 pts)** Use several diagrams to briefly explain its main idea of using hybrid hash join to do R JOIN S;

1) Partition R0 of table R is kept in the memory after the partitioning phase.

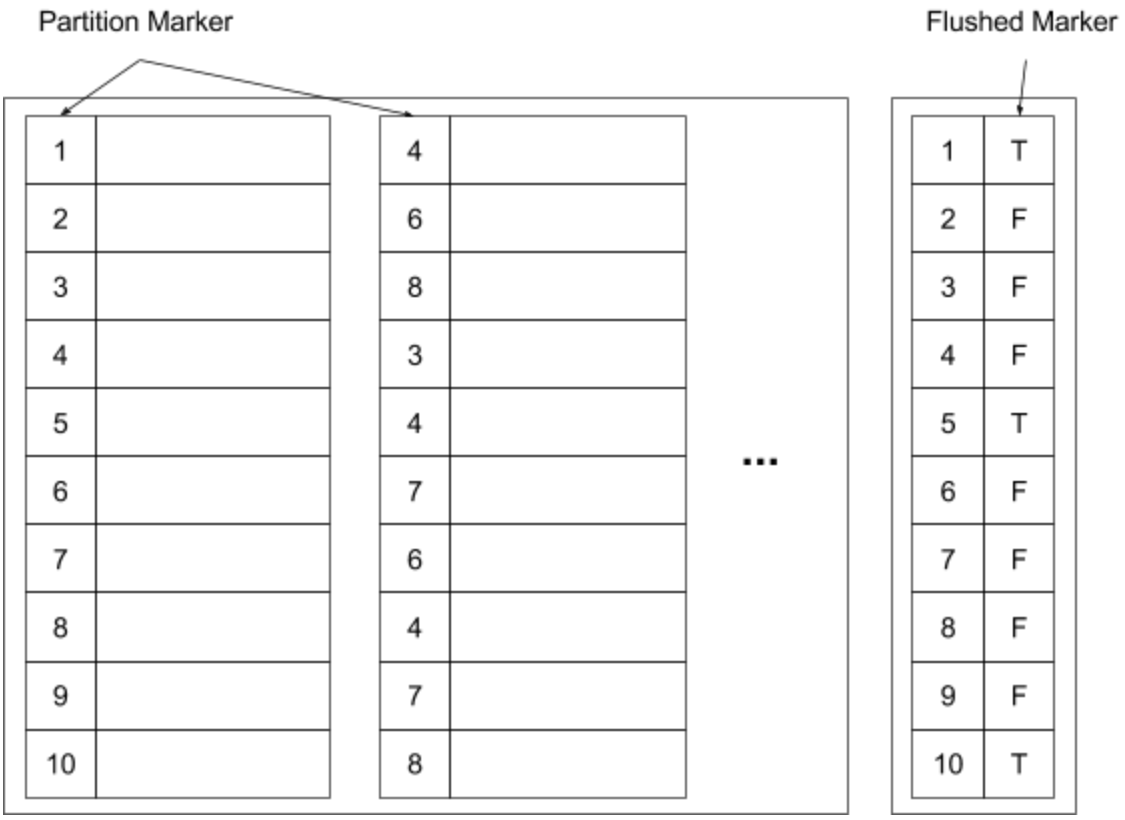



2) Build in-memory hash table from R0 directly. During partitioning phase of S, we can do the matching of R0 and S0 at the same time. The matching process of the remaining partitions is the same with normal Grace Hash Join.




b) **(9 pts)** In the build phase, the algorithm keeps one of the R partitions called "R0" in memory to avoid disk IOs for the blocks in this partition. Very often it's very hard to have a good hash function to generate R0 such as its size fits to the available memory (after using other buffers for the other partitions). Come up with an idea to improve this hybrid hash join so that it can work for any given hash function. In other words, the input of the algorithm is the number of available memory buffers B, a hash function H, and relations R and S. To make your description more clear, let's assume we have B = 500 buffer pages in memory, and we are given a hash function *h* that generates 10 partitions. Your new algorithm can be "adaptive" so that it can keep as many R partitions in memory as possible. **Hint**: more than one partition can be kept in memory.

Note: (1) Make sure your answer is succinct and clear; and (2) You are encouraged to use diagrams to explain the idea.




Each partition is assigned a buffer page at the beginning. When a page is full, we assign the partition another buffer page if possible instead of flushing the partition out directly. If there is no available buffer page, choose one of the partitions to be flushed out. Use a data structure to keep track of which partitions have already been flushed out and do not assign them extra buffer pages in the future. When the partitioning of R is finished, use the partitions that have never been flushed out to build the in-memory hash table for the matching phase.